IEEE TRANSACTIONS ON
EMERGING TOPICS
IN COMPUTING

Received 21 August 2019; revised 30 March 2020; accepted 30 March 2020.
Date of publication 22 April 2020; date of current version 4 March 2022.

*Digital Object Identifier 10.1109/TETC.2020.2986487*

# Incremental Checkpointing for Fault-Tolerant Stream Processing Systems: A Data Structure Approach

**CHIA-YU LIN, (Member, IEEE), LI-CHUN WANG, (Fellow, IEEE), AND SHU-PING CHANG, (Member, IEEE)**

C.-Y Lin and L.-C Wang are with the Department of Electrical and Computer Engineering, National Chiao Tung University, Hsinchu 300, Taiwan
S.-P. Chang is with the IBM T.J. Watson Research Center, Yorktown Heights NY 10598, USA
CORRESPONDING AUTHOR: L.-C. WANG (Email: lichun@g2.nctu.edu.tw)

**ABSTRACT** As the demand of high-speed stream processing grows, in-memory databases are widely used to analyze streaming data. It is challenging for in-memory systems to meet the requirements of high throughput and data persistence at the same time since data are not stored in disks. ARIES logging and command logging are two popular logging methods. In current applications, both ARIES logging and command logging are necessary. However, no checkpointing mechanism includes both the functions of ARIES logging method and command logging method. Besides, adopting ARIES logging method in an in-memory database creates high overhead. Command logging records redundant commands and has high storage cost. To address the above issues, we utilize order-irrelevant characteristics of data structure and incremental checkpointing concepts to devise a data structure based incremental checkpointing (DSIC) mechanism. DSIC mechanism is a very low overhead checkpointing approach while retaining the features of ARIES logging and command logging. DSIC mechanism reduces more than 70 percent logging time of the existing logging scheme and saves 40 percent storage costs of the existing logging scheme.

**INDEX TERMS** Stream processing systems, incremental checkpointing, in-memory databases, key-value stores

## I. INTRODUCTION

Streaming data technology has generated great interest due to the growing demands for rapid analysis. Streaming data are collected in real-time through sensors, GPS, social media activities, and e-Commerce, etc. Deriving accurate insights in the context in which these streaming data are created can exceed the market share that your competitors have not yet identified. Hence, real-time streaming data analysis is expected to create new services and improve decision making [1], [2].

To achieve real-time analysis, streaming data are analyzed and stored in the in-memory systems [3]. For example, autonomous cars collect and analyze traffic data in the in-memory system to provide passengers an efficient and safer transportation environment [4], [5]. The route of autonomous electric buses are recorded in memory to predict power-saving routing and charging locations in real-time [6]. Since data are not stored in disks, it is challenging for in-memory systems to meet the requirements of high throughput and data persistence at the same time. Fault tolerance is critical when streaming operators crash and result in data loss [7], [8], [9]. Snapshot and log-based methods are common data persistence and recovery models for in-memory databases [10]. Compared to the snapshot approaches, log-based methods are more popular. In [11], ARIES logging method recorded how transactions update tuples and was the most widely used logging method. High overhead was incurred by ARIES logging method since all the transactions were processed rapidly in the in-memory databases. To reduce the logging overhead, a command logging method [12] was proposed to only record the transactions. Command logging resulted in low recovery performance since it fully replayed transactions and lacked parallel recovery in distributed systems [13]. Thus, [14] proposed a distributed command logging method by generating a dependency graph to identify the dependency relationship between commands. However, many unnecessary commands were still logged in [14] and had big storage cost. For example, adding $item_i$ to a set and removing $item_i$ from the same set could be offset. Adding $item_i$ twice to a set could be regarded as one ADD operation. The redundant commands intensively increased the recovery time and storage cost of command logging method.

In current applications, both ARIES logging and command logging are necessary. For example, autonomous electrical

buses need ARIES logging to retrieve the latest status of bus. On the other hand, autonomous electrical buses need to replay the routing events to find out the reasons of the accident. However, no checkpointing mechanism includes both the features of ARIES logging method and command logging method. In addition, the high overhead of ARIES logging method and the redundant commands of command logging method have a great impact on system performance.

In this paper, we propose a data structure based incremental checkpointing (DSIC) mechanism to provide an efficient checkpointing process while preserving the functionalities of ARIES logging method and command logging method for in-memory databases. In DSIC mechanism, we utilize the order-irrelevant characteristics of sets to check the validity of commands and remove redundant commands. The storage cost and logging time of DSIC mechanism can be intensively reduced. Besides, an undo incremental checkpointing on set scheme (DSIC-undo) and a redo incremental checkpointing on set scheme (DSIC-redo) are designed to retrieve the latest status in real-time and to replay commands. DSIC-undo and DSIC-redo schemes can be switched between each other and replay to a specific state.

The contributions of DSIC mechanism are as follows:

- DSIC mechanism reduces more than 70 percent logging time of the existing logging scheme.
- DSIC mechanism saves 40 percent storage costs of the existing logging scheme.
- DSIC mechanism is designed based on the existing key-value store application programming interfaces (API) to address the hardware support issues. The proposed techniques do not require changes in database codes.
- The checkpointing techniques of DSIC mechanism can be adaptive for read- and write-intensive streaming applications.

The rest of this paper is organized as follows. Section II describes the related works for checkpointing techniques. Section III details the proposed data structure based incremental checkpointing (DSIC) mechanism. Section IV demonstrates the experimental results of DSIC mechanism. We give our concluding remarks in Section V.

## II. RELATED WORK

In stream processing systems, different fault tolerance techniques have been proposed in the past to solve the challenges of data reliability and short recovery time [15], [16]. In [17], the authors logged the entire operator state of streaming applications into Distributed File System (DFS) such as Google File System (GFS) and Hadoop Distributed File System (HDFS). [18] logged updated data and periodically generated checkpoints to reduce the size of the log file. When a failure occurred, the latest checkpoint state was loaded and the logs were replayed. [19] split the operator states into multiple parts to incrementally update operator states. To dynamically scale out and restore stateful operators, [20] performed the explicit operator state management by externalizing internal operator states. Most studies only focused on replicating operator states on disks. In-memory databases are now commonly adopted as auxiliary tools in streaming applications. Streaming data are written into in-memory databases when multiple operators share states [21] or the state of operators is big. Thus, the replication of in-memory databases with operator states becomes important.

Data replication and data persistence were two methods to keep data safe for in-memory databases such as Redis [22], [23], Memcached [24], MongoDB [25] and so on. [26], [27], [28], [29], [30] adopted data replication techniques to achieve fault tolerance. In [26], a standby replicated node would take over the role of the primary node in the event of a failure. [27] partitioned Redis data of the master server to slave servers. The recovery process efficiently replicated data from various slave servers. [28] designed a master-slave semi synchronization scheme. The scheme recorded the full replication of data on slave nodes and reduced the influence on the performance of Redis by TCP protocol. [29] developed a value identifier vector and a dictionary log for column-oriented in-memory databases. A transaction with the corresponding column of the in-memory database was added to the value identifier vector and the dictionary log of the slave node. [30] proposed MemEC, which was an erasure-coding based key-value store. MemEC encoded objects including keys, values, and metadata through a new fully encoded data model to achieve low data access latency and fast recovery from in-memory storage servers. In data replication technology, computational performance and reliability were trade-offs since it took a long time to fully replicate data.

Snapshots and log-based approaches were two data persistence models [10]. Based on snapshots and log-based approaches, Redis designed RDB persistence and AOF persistence. RDB persistence executed point-in-time snapshots of dataset. AOF persistence logged each WRITE operation received by the server [31]. To avoid data loss, [32] often wrote RDB to storage class memory (SCM) and recorded updated items in SCM via AOF file. In contrast, log-based approaches were more popular than snapshot approaches. The most widely used logging method was ARIES logging method [11]. ARIES logging method recorded how transactions updated tuples. [33] used a sequential log in persistent storage to maintain checkpoints for key-value pairs. However, adopting ARIES logging method in an in-memory database created high overhead because all transactions are processed quickly in memory.

To reduce logging overhead, a command logging method [12] was proposed to only record the transactions. Compared to ARIES logging method, the command logging method compressed log information to reduce the I/O cost of storing logs into disks. However, in recovering process, command logging method seriously fully replayed transactions instead of executing in parallel and resulted in long recovery time. To improve the performance of command logging, [14] proposed a distributed command logging method by identifying

**TABLE 1. Notations of DSIC-undo and DSIC-redo schemes.**

| Notations | Meaning |
|---|---|
| $T$ | Current state. |
| $T-1$ | The previous state. |
| $K$ | The key of the set. |
| $M$ | The number of members we add/remove to the set. |
| $(K, member_{1...M})$ | The key-value pair we add/remove to the set. |
| $K_B$ | The key of the *base set*. |
| $K_{A\_T}$ | The key of the *add set* at state $T$. |
| $K_{R\_T}$ | The key of the *remove set* at state $T$. |
| $S_B$ | The *base set*. |
| $S_{A\_T}$ | The *add set* of state $T$. |
| $S_{R\_T}$ | The *remove set* of state $T$. |
| $N$ | The number of ADD/REMOVE operations between two checkpoints. |
| $Size_B$ | Size of *base set*. |

the dependencies between commands. However, since many redundant commands were still logged and replayed in [14], the cost of recovery process significantly increased.

Current applications need ARIES logging method and command logging method at the same time. For example, autonomous electrical buses can adopt ARIES logging to retrieve the latest status of bus and provide real-time decisions. They can also utilize command logging method to find out the reasons of an accident by replaying the routing events. However, there is no checkpointing mechanism which includes both functions of ARIES logging method and command logging method. In addition, ARIES logging has high overhead. Command logging records redundant commands and has high storage cost.

In this paper, we want to improve the performance of checkpointing while retaining the functionalities of both ARIES logging and command logging. We propose a data structure based incremental checkpointing (DSIC) mechanism by utilizing the features of data structures such as set and incremental checkpointing techniques [34], [35], which can only log changed data, to reduce the amount of data in each checkpoint. The detail of DSIC mechanism will be introduced in the following sections.

## III. DATA STRUCTURE BASED INCREMENTAL CHECKPOINTING MECHANISM

In the proposed data structure based incremental checkpointing (DSIC) mechanism, the metadata file, checkpoint logs and the *base set* are three basic components defined in the in-memory database, which is a key-value store. In the metadata file, a mapping file between the original key and the extended key is stored. In checkpoint logs, the *add sets* and the *remove sets* are generated to record the added and removed value in each state, respectively. The *base set* records all members in the set. If the *base set* is updated in each operation and the latest *base set* is loaded to "undo" the operations in the rollback process, it is DSIC-undo scheme. Since the maintenance of the latest *base set* makes the storage requirements

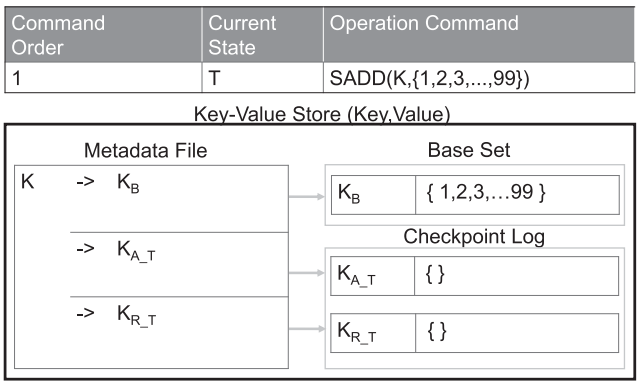



**FIGURE 1. An example of creating the *base set* in DSIC-undo scheme.**

for DSIC-undo scheme increase, DSIC-redo scheme is proposed. In each operation, the system only updates the modified values in the *add set* and the *remove set*. During the rollback procedure, the system "re-does" the operations on the *base set*.

The most common set operations as following are demonstrated in DSIC-undo and DSIC-redo schemes:

- SADD: Add one or more members to the set of the key.
- SREM: Remove one or more members from the set of the key.
- SISMEMBER: Determine whether a given value is a member of the set of the key.
- SCARD: Retrieve the number of members in the set of the key.
- CHECKPOINT: Change the state of the store.
- ROLLBACK: Restore the data to a previous state.

Notations of DSIC-undo and DSIC-redo schemes are shown in Table 1. In the initial step of state $T$, the mapping between the new key ($K$), the key of the *base set* ($K_B$), the key of the *add set* ($K_{A\_T}$), and the key of the *remove set* ($K_{R\_T}$) are created in the metadata file. The *base set* ($S_B$), the *add set* ($S_{A\_T}$) and the *remove set* ($S_{R\_T}$) of state $T$ are also generated in checkpoint logs. When the members are initially added to the set of key $K$, the members are added to $S_B$, as shown in Figure 1.

### A. DSIC-UNDO SCHEME

In DSIC-undo scheme, the system not only logs the operations in the *add set* and the *remove set*, but also updates the *base set*.

#### 1) SADD

Algorithm 1 shows the process of adding $(K, member_{1...M})$ in DSIC-undo scheme. A loop is used to update $member_{1...M}$ one by one in $S_B$, $S_{A\_T}$, and $S_{R\_T}$ of key $K$. In the loop, $member_i$ is added to $S_B$. If the member is not in $S_B$ before SADD operation, $S_{R\_T}$ is used to check whether the member is in it. If the member is in $S_{R\_T}$, the member is removed from $S_{R\_T}$. Otherwise, the member is added to $S_{A\_T}$. The time complexity of an add operation in the in-memory database is

| Command Order | Current State | Operation Command | The process in key-value store |
|---|---|---|---|
| 1 | T | SADD(K,100) | ❶❷ |
| 2 | | SADD(K,100) | |
| 3 | | SREM(K,99) | ❸❹ |
| 4 | | SADD(K,99) | ❺❻ |

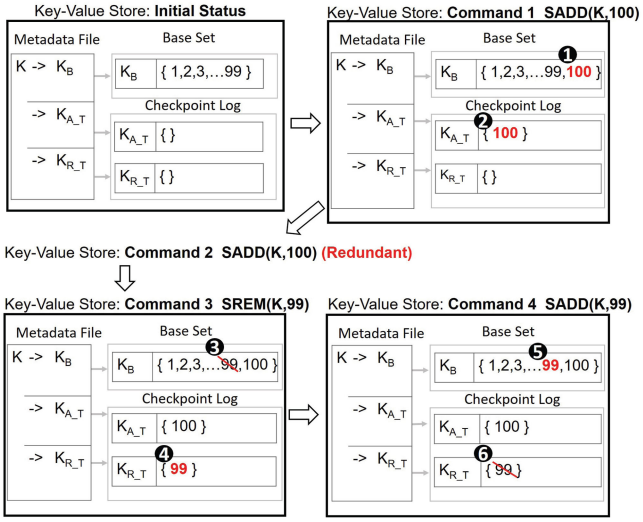



**FIGURE 2. An example of executing SADD and SREM in DSIC-undo scheme.**

$O(N)$, where $N$ is the number of members added to the set. Therefore, the time complexity of a SADD operation is $O(N) + O(N) = O(N)$.

**Algorithm 1.** SADD of DSIC-Undo Scheme

**procedure** SADD($K, member_{1...M}$)
1: $i \leftarrow 1$
2: **for** $i < M + 1$ **do**
3: Add $member_i$ to $S_B$
4: $r \leftarrow$ return value of ADD operation to *base set*
5: **if** $r == 1$ **then**
6: **if** $member_i \in S_{R\_T}$ **do**
7: Remove $member_i$ from $S_{R\_T}$
8: **else**
9: Add $member_i$ to $S_{A\_T}$
10: **end if**
11: **end if**
12: $i \leftarrow i + 1$
13: **end for**
**end procedure**

### 2) SREM

The SREM procedure is similar to the SADD procedure, as shown in Algorithm 2. The $member_i$ is first removed from $S_B$. If the $member_i$ is in $S_B$ before the SREM operation, the $member_i$ is removed from $S_{A\_T}$ or added to $S_{R\_T}$. The time complexity of a remove operation in the in-memory database is $O(N)$, where $N$ is the number of members removed from the set. Therefore, the time complexity of a SREM operation is $O(N) + O(N) = O(N)$.

Figure 2 shows the examples of SADD and SREM operations. Initially, $S_B$ contains $1, 2, 3, ...99$. When the first

| Command Order | Current State | Operation Command | Return Value |
|---|---|---|---|
| 1 | T+1 | SISMEMBER(K,99) | FALSE |
| 2 | | SCARD(K) | 100 |

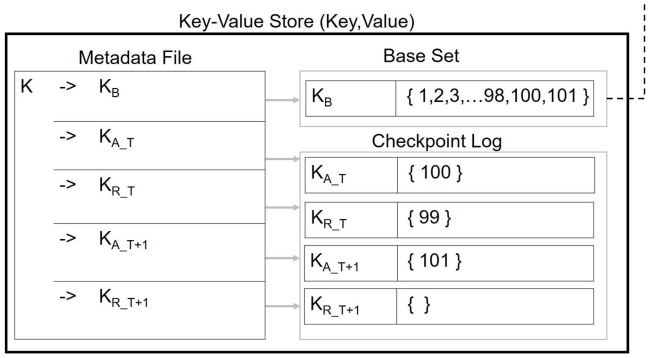



**FIGURE 3. An example of executing SISMEMBER and SCARD in DSIC-undo scheme.**

$SADD(K, 100)$ is requested, 100 is added to the *base set* and the *add set*. In the second command, 100 is added again. Since 100 is already in the *base set*, the updating processes in the *remove set* and the *add set* are not executed. 99 is requested to remove from the set in the third command. 99 is removed from the *base set* and added to the *remove set*. When $SADD(K, 99)$ is executed, 99 is added to the *base set* and removed from the *remove set*.

**Algorithm 2.** SREM of DSIC-Undo Scheme

**procedure** SREM($K, member_{1...M}$)
1: $i \leftarrow 1$
2: **for** $i < M + 1$ **do**
3: Remove $member_i$ from $S_B$
4: $r \leftarrow$ return value of *base set* REMOVE operation
5: **if** $r == 1$ **then**
6: **if** $member_i \in S_{A\_T}$ **then**
7: Remove $member_i$ from $S_{A\_T}$
8: **else**
9: Add $member_i$ to $S_{R\_T}$
10: **end if**
11: **end if**
12: $i \leftarrow i + 1$
13: **end for**
**end procedure**

### 3) SISMEMBER AND SCARD

In addition to ADD/REMOVE operations, READ operations are common. The READ operation of set is using "SISMEMBER" to read an element or using "SCARD" to get the size of a set. Since the *base set* is maintained as the latest set, SISMEMBER and SCARD can directly utilize *base set* to get the information, as shown in Figure 3. When 99 is read in state $T + 1$, the *base set* is checked. Since 99 is not in $S_B$, "FALSE" is returned to the user. The time complexity of a read operation in the in-memory database is $O(1)$. Therefore, the time complexity of using the *base set* to finish a SISMEMBER or a SCARD operation is $O(1)$.

| Command Order | Current State | Operation Command | The process in key-value store |
|---|---|---|---|
| 1 | T | SADD(K,100) | |
| 2 | | SREM(K,99) | |
| **3** | | **Checkpoint** | ❶❷ |
| 4 | **T+1** | | |

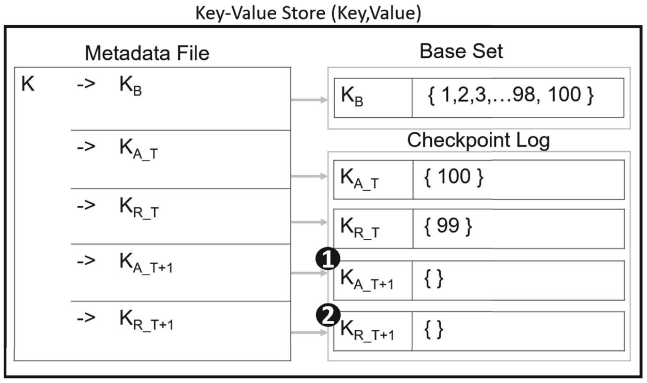



**FIGURE 4. An example of executing checkpoint in DSIC-undo scheme.**

#### 4) CHECKPOINT

In state $T$, SADD and SREM are executed. The *base set* is updated in the SADD and SREM procedures. When a checkpoint is established, the state becomes $T+1$. The *add set* and the *remove set* of state $T+1$ are created, as shown in Figure 4. The *base set* remains the same as the previous step. The time complexity of CHECKPOINT process is $O(1)$.

**Algorithm 3.** ROLLBACK of DSIC-Undo Scheme

**procedure** `Rollback`
1: $member[] \leftarrow$ The members of $S_{A\_T+1}$
2: **for** $i <$ The size of $S_{A\_T+1}$ **do**
3: Remove $member[i]$ from $S_B$
4: **end for**
5: $member[] \leftarrow$ The members of $S_{R\_T+1}$
6: **for** $i <$ The size of $S_{R\_T+1}$ **do**
7: Add $member[i]$ from $S_B$
8: **end for**
9: Delete $S_{A\_T+1}$
10: Delete $S_{R\_T+1}$
11: current state $\leftarrow T$
**end procedure**

#### 5) ROLLBACK

In a system failure event, we can "undo" the operations by checkpoint logs. Algorithm 3 shows the four steps of the rollback procedure from state $T+1$ to state $T$. The first step is to remove the members of $S_{A\_T+1}$ from $S_B$. The time complexity is $O(N)$, where $N$ is the number of members removed from the set. 99 is removed from the *base set* in Figure 5. The members of $S_{R\_T+1}$ are added to $S_B$ in the second step. The time complexity is $O(N)$, where $N$ is the number of members added to the set. $S_{A\_T+1}$ and $S_{R\_T+1}$ are deleted in the third step. The time complexity is $O(N)$, where $N$ is the number of

| Command Order | Current State | Operation Command | The process in key-value store |
|---|---|---|---|
| 1 | T | SADD(K,100) | |
| 2 | | SREM(K,99) | |
| **3** | | **Checkpoint** | |
| 4 | T+1 | SADD(K,99) | |
| **6** | | **Rollback** | ❶❷❸ |
| 7 | T | | |

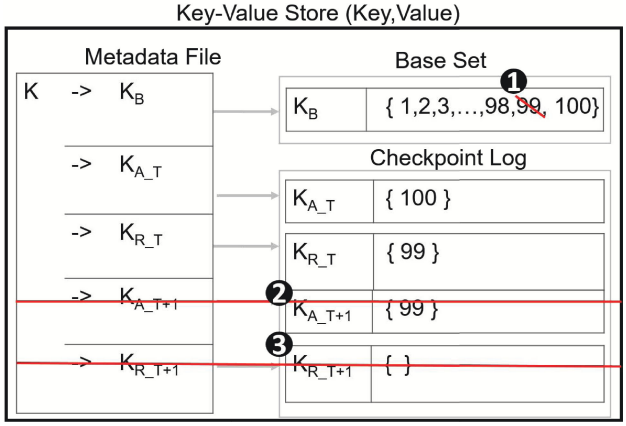



**FIGURE 5. An example of executing ROLLBACK in DSIC-undo scheme.**

sets to delete. Finally, the current state changes to $T$. After the undo operations is completed, the *base set* is the latest version in state $T$. The total time complexity of a rollback operation is $O(N)+O(N)+O(N)=O(N)$.

Since the *base set* is maintained as the latest set, the READ operation such as SISMEMBER and SCARD can be directly operated on the *base set*, thereby reducing the computational complexity of the READ operation. That is, DSIC-undo scheme is well-suited for read-intensive streaming applications. However, maintaining large and latest *base set* requires not only additional storage but also an additional step to update checkpoint logs. The checkpoint time and storage costs increase. Therefore, DSIC-redo scheme is proposed to reduce the storage cost and the complexity of checkpointing process.

### B. DSIC-REDO SCHEME

In DSIC-redo scheme, the *base set* contains the initial members of the current state instead of being updated in every operation. Since the members in the *base set* are not the latest members, we can not check whether a member is in the set or not. Therefore, we must record every modified member in both *add set* and the *remove set*.

#### 1) SADD

Algorithm 4 is the SADD procedure. The added members are removed from $S_{R\_T}$ and added to $S_{A\_T}$. The time complexity of an add/remove operation in the in-memory database is $O(N)$, where $N$ is the number of members added to/removed from the set. Therefore, the time complexity of a SADD operation is $O(N)+O(N)=O(N)$.

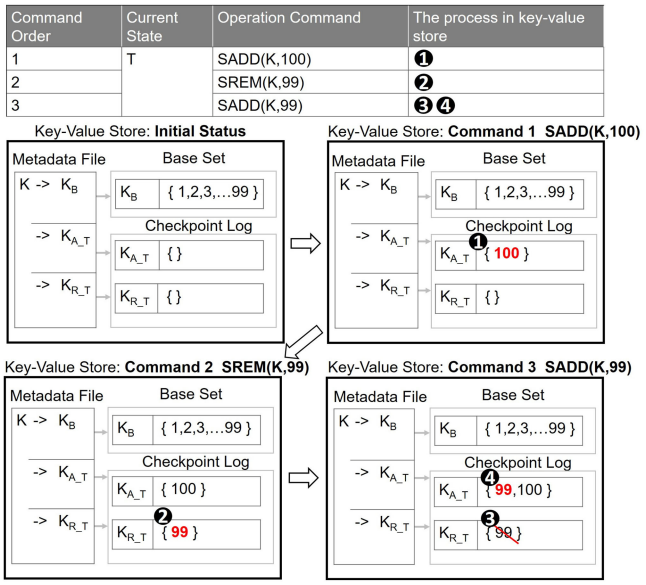



**FIGURE 6. An example of executing SADD and SREM in DSIC-redo scheme.**

**Algorithm 4.** SADD of DSIC-Redo Scheme

**procedure** SADD($K, member_{1...M}$)
1: $i \leftarrow 1$
2: **for** $i < M + 1$ **do**
3: **if** $member_i \in S_{R\_T}$ **then**
4: Remove $member_i$ from $S_{R\_T}$
5: **end if**
6: Add $member_i$ to $S_{A\_T}$
7: $i \leftarrow i + 1$
8: **end for**
**end procedure**

**Algorithm 5.** SREM of DSIC-Redo Scheme

**procedure** SREM($K, member_{1...M}$)
1: $i \leftarrow 1$
2: **for** $i < M + 1$ **do**
3: **if** $member_i \in S_{A\_T}$ **then**
4: Remove $member_i$ from $S_{A\_T}$
5: **end if**
6: Add $member_i$ to $S_{R\_T}$
7: $i \leftarrow i + 1$
8: **end for**
**end procedure**

### 2) SREM

Algorithm 5 presents the SREM procedure. When the SREM operation is performed, the members are removed from $S_{A\_T}$ and added to $S_{R\_T}$. The time complexity of an add/remove operation in the in-memory database is $O(N)$, where $N$ is the number of members added to/removed from the set. Therefore, the time complexity of a SREM operation is $O(N) + O(N) = O(N)$.

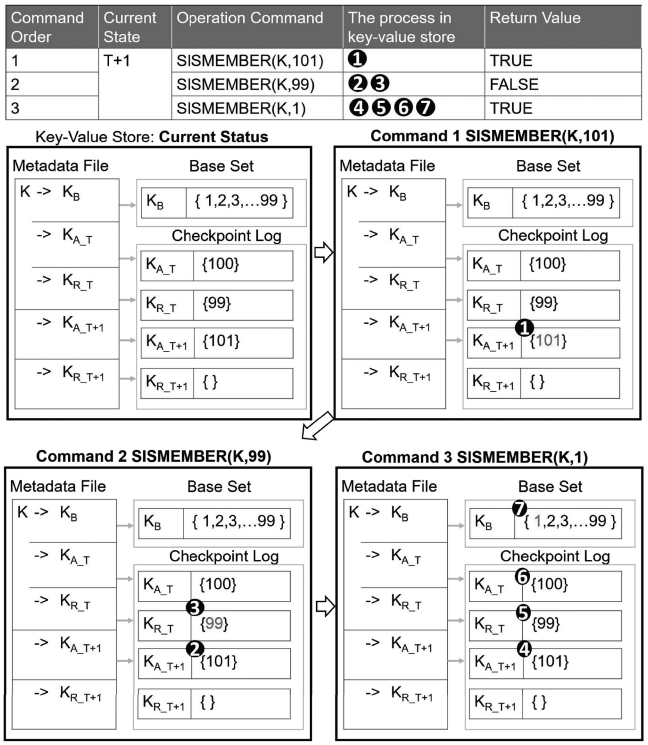



**FIGURE 7. An example of executing SISMEMBER in DSIC-redo scheme.**

Figure 6 is an example of the SADD and SREM process. In the first command, 100 is added to the *add set*. When 99 is requested to remove from the set of key $K$, 99 is added to the *remove set*. When 99 is added again to the set, 99 is removed from the *remove set* and added to the *add set*.

### 3) SISMEMBER

Algorithm 6 shows SISMEMBER process. Since the *base set* contains the initial members of the current state, we have to check *remove set* ($S_{R\_T}$), *add set* ($S_{A\_T}$) *and base set* ($S_B$) in order to see if the member is in the set or not. $S_{R\_T}$ records the members removed from the set of key $K$ in state $T$. Thus, $S_{R\_T}$ is checked first. If the specified member is in $S_{R\_T}$, "FALSE" is returned to the user. Otherwise, $S_{A\_T}$ is checked. $S_{A\_T}$ records the members added to the set of key $K$ in state $T$. If the member is in $S_{A\_T}$, "TRUE" is returned to user. If the value is neither in $S_{A\_T}$ nor $S_{R\_T}$ of state $T$, we check the *add set* and the *remove set* of state $T - 1$ to state 1. If we still can not find the member in the *add set* and the *remove set* of any states, we will check $S_B$. "TRUE" is returned to the user, if the member is found in $S_B$; otherwise, it returns "FALSE." The time complexity of a read operation in the in-memory database is $O(1)$. Therefore, the time complexity of a SISMEMBER operation is $C * O(1)$, where $C$ is the number of sets we need to check.

Figure 7 is an example of SISMEMBER procedure. 101 is found in the *add set* of state $T + 1$, and returns "TRUE." 99 is not in the *add set* and the *remove set* of state $T + 1$, but 99 is in the *remove set* of state $T$. Thus, 99 is removed in state $T + 1$. "FALSE" is returned to the user. 1 is not found in the


**FIGURE 8.** Two methods of rollback procedure.

*add set* and the *remove set* of any states but 1 is found in the *base set*. That is, 1 is in the set of key $K$.

**Algorithm 6.** SISMEMBER of DSIC-Redo Scheme

**function** SISMEMBER($K$, *member*)
1: $Q \leftarrow$ T
2: **for** $Q > 0$ **do**
3: **if** *member* $\in S_{R\_Q}$ **then**
4: **return** FALSE
5: **end if**
6: **if** *member* $\in S_{A\_Q}$ **then**
7: **return** TRUE
8: **end if**
9: $Q \leftarrow Q - 1$
10: **end for**
11: **if** *member* $\in S_B$ **then**
12: **return** TRUE
13: **else**
14: **return** FALSE
15: **end if**
**end function**

#### 4) CHECKPOINT

The CHECKPOINT operation of DSIC-redo scheme is same as that of the DSIC-undo scheme. The state becomes $T + 1$ after a checkpoint. The *add set* and the *remove set* of state $T + 1$ are created.

#### 5) SCARD

In addition to SISMEMBER process, SCARD operation, which finds the size of the set, is also complex. Since we do not maintain the latest *base set*, the members of the *add set* might be a duplicate member in the *base set*. We can not simply summarize the size of the *add sets* and subtract the size of the *remove sets* to get the answer. We must combine the *base set* with the *add set* and the *remove set* of every state to generate a *new base set*. The size of *new base set* is replied to the user. The combination step is same as rollback step, which is described in detail in the next subsection.

#### 6) ROLLBACK

In DSIC-redo mechanism, we record the add set and the remove set of every state. Thus, we can choose to rollback to

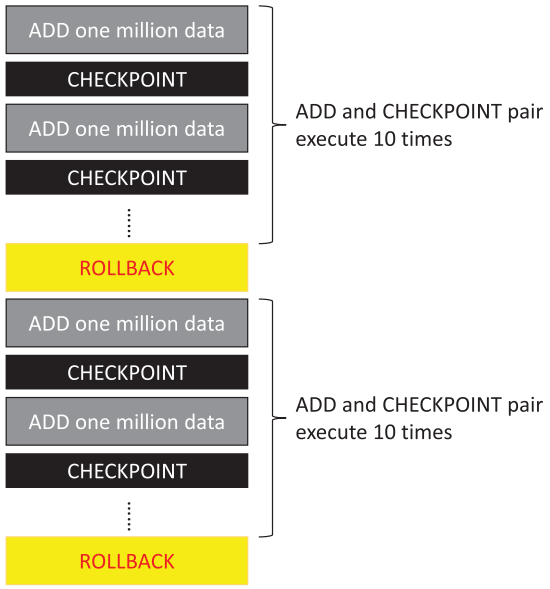



**FIGURE 9.** Experiment steps for comparing execution time of different ROLLBACK methods.

specific state. Assume we want to rollback to state $P$, which is a state before current state $T$. We calculate the union of the *base set* with the *add sets* from state $P, P + 1, P + 2...$ to state $T$. After generating the new union set, we determine the difference between the new union set and the *remove sets* from state $P$ to state $T$.

There are two methods to calculate the latest *base set*. In method 1, the *base set* is used in each step to calculate the union, as shown in Figure 8(a). Suppose the checkpoint logs contain the log from *state* 1 to *state* 4. There are 20 million members in the *base set* and there are one million members in each *add set*. At first, we calculate the union of the *add set* of *state* 1 and the *base set* to get the new *base set* $S_{B\_new}$. There are 21 million members in $Set_{B\_new}$. In the following steps, we use $Set_{B\_new}$ to get the union set with the *add sets* of *state* 2 to *state* 4. In method 1, the total number of members processed in the union operation is 90 million. In method 2, *add sets* of each state are used to calculate the new union set. In the final step, the latest new union set is calculated with the *base set* to get the final union set, as shown in Figure 8(b). The total number of members processed in the union operations of method 2 is 32 million. Since the time complexity of union operation is $O(N)$, where $N$ is the total number of elements in all given sets, the time complexity of method 2 is much lower than method 1.

We also conduct an experiment to compare the execution time of methods 1 and method 2. Figure 9 illustrates the steps of the experiment. First, we add one million members to the set and generate a checkpoint after the ADD operation. A rollback procedure is operated after the ADD operation and the CHECKPOINT operation are repeated ten times. We execute rollback procedure two times to measure the execution time. In Figure 10, we can see that the execution time of method 1 in the first rollback process is about 1.8 times

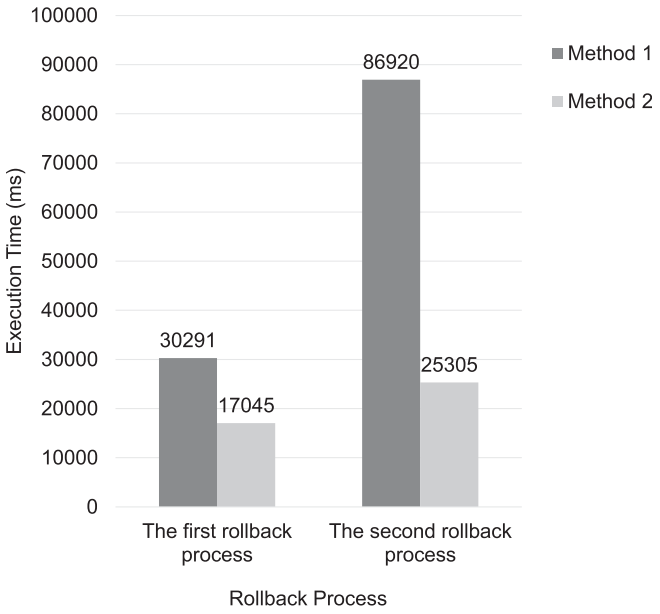



**FIGURE 10.** **Execution time of different ROLLBACK methods.**

execution time of method 2. In the second rollback process, the execution time of method 1 is about 4 times execution time of method 2. That is, if the system executes rollback processes multiple times, method 2 can achieve better performance. Method 2 is chosen to implement in the rollback procedure of DSIC-redo scheme, as shown in Algorithm 7. The time complexity of a union/difference operation is $O(N)$, where $N$ is the total number of elements in all given sets. Therefore, the time complexity of a rollback operation is $C * O(N)$, $C$ is the number of union and difference operations.

**Algorithm 7.** ROLLBACK of DSIC-Redo Scheme

1: $P \leftarrow$ The earliest state $(P \neq T-1)$
2: The state of the checkpoint log records from $P$ to $T$
**procedure** `Combine`
3: $S_{A\_new} \leftarrow S_{A\_P}$
4: $S_{R\_new} \leftarrow S_{R\_P}$
5: **while** $S_{A\_new}$ does not contain all members of $S_A$ from $P$ to $T-1$ **do**
6: $S_{A\_new} \leftarrow$ The union of all $S_A$ from $P$ to $T-1$
7: Delete $S_A$ from $P$ to $T-1$
8: **end while**
9: **while** $S_{R\_new}$ does not contain all members of $S_R$ from $P$ to $T-1$ **do**
10: $S_{R\_new} \leftarrow$ The union of all $S_R$ from $P$ to $T-1$
11: Delete $S_R$ from $P$ to $T-1$
12: **end while**
13: $S_{B\_new} \leftarrow$ The union of $S_B$ and $S_{A\_new}$
14: $S_{B\_new} \leftarrow$ The difference between $S_{B\_new}$ and $S_{R\_new}$
15: $S_B \leftarrow S_{B\_new}$
**end procedure**

Figure 11 illustrates the ROLLBACK operation. The rollback is from state $T+3$ to state $T+1$. We calculate the

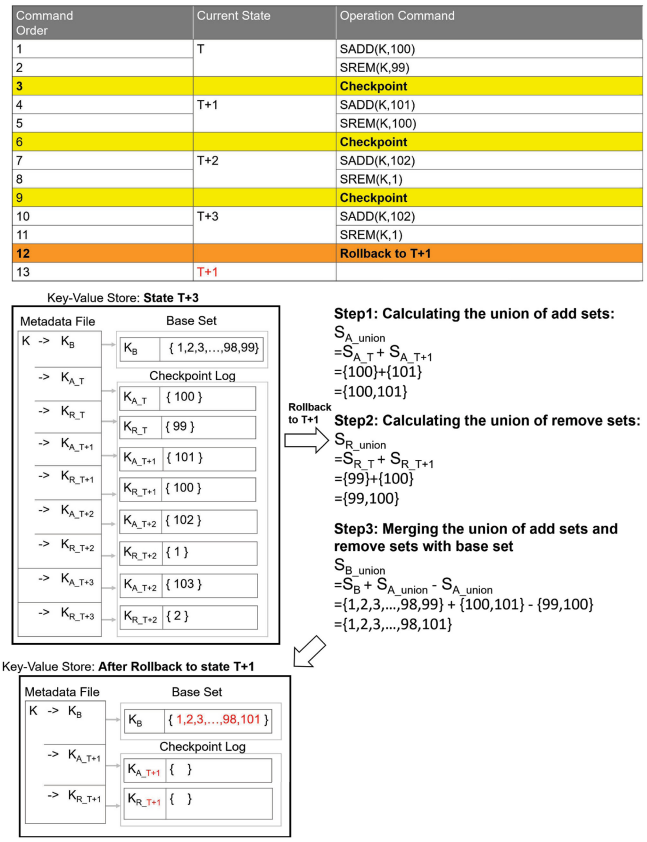



**FIGURE 11.** **An example of executing ROLLBACK in DSIC-redo scheme.**

union of the *add sets* of state $T$ and state $T+1$ is $\{100, 101\}$. The union of the *remove sets* of state $T$ and state $T+1$ is $\{99, 100\}$. We calculate the union of the *base set* $\{1, 2, 3, \ldots, 98, 99\}$ and the new union of *add sets* $\{100, 101\}$ to get $\{1, 2, 3, \ldots, 98, 99, 100, 101\}$. We compare the difference of $\{1, 2, 3, \ldots, 98, 99, 100, 101\}$ and the new union of *remove sets* $\{99, 100\}$. In the end, the latest *base set* is $\{1, 2, 3, \ldots, 98, 101\}$. Since the *base set* contains the latest members of state $T+1$, the *add set* and the *remove set* in the checkpoint logs are deleted.

## IV. EVALUATION AND DISCUSSION

In the experiments, we use Redis, a popular high-performance memory oriented key value database. We set up two machines: one is for the Redis server and one is for the client. Each server has 4 cores, 64 GB memory, 256 GB solid-state disk and 1 TB local disks. We compare the proposed DSIC mechanism with ARIES logging approach [11] and command logging approach [14]. ARIES logging approach records how database updated by commands, which is called "ARIES" in the following experiments. Command logging approach logs the requested command, which is called "COMMAND" in the following experiments. The execution time and storage costs of the ADD/REMOVE operation, the READ operations, and the ROLLBACK operations are evaluated in the following experiments.

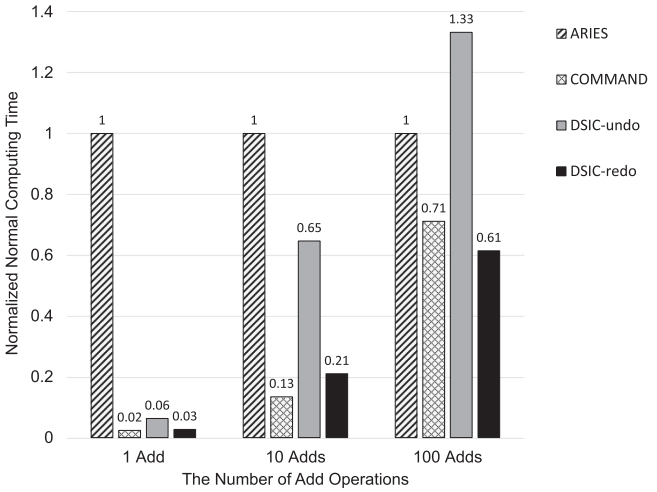



**FIGURE 12. Normalized normal computing time of various number of ADD operations between two checkpoints.**

### A. EXPERIMENT OF ADD/REMOVE AND CHECKPOINT OPERATIONS

To measure the performance of ADD/REMOVE operations and CHECKPOINT operations, the normal computing time is considered. The normal computing time represents the average computation time of one ADD/REMOVE operation and one CHECKPOINT operation as follows:

$$\text{Normal computing time} = \frac{\text{Time of ADD/REMOVE operations} + \text{Time of one CHECKPOINT operation}}{\text{Number of ADD/REMOVE operations}}. \quad (1)$$

The number of ADD/REMOVE operations between two checkpoints ($N$), the number of members added/removed to/from the set ($M$), and the size of the *base set* ($Size_B$) are three factors that affect the performance of ADD/REMOVE operations and CHECKPOINT operations. The following experiment shows the impact of three factors.

#### 1) NUMBER OF ADD/REMOVE OPERATIONS BETWEEN TWO CHECKPOINTS ($N$)

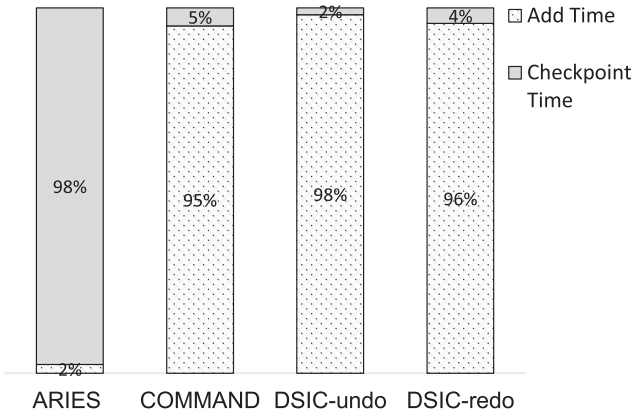
In this experiment, $N$ is set to one, ten and one hundred to measure the normal computing time. $Size_B$ is one million members in Redis, and $M$ is ten thousand members. If $N$ is one, the normal computing time of ARIES, COMMAND, DSIC-undo, and DSIC-redo schemes are 374 ms, 9 ms, 23 ms and 10 ms respectively. When $N$ is ten, the normal computing time of ARIES, COMMAND, DSIC-undo, and DSIC-redo schemes are 53 ms, 7 ms, 34 ms, and 11 ms respectively. When $N$ is one hundred, the normal computing time of ARIES, COMMAND, DSIC-undo, and DSIC-redo schemes are 10 ms, 7 ms, 13 ms, and 6 ms respectively. The normal computing time of four approaches is normalized by the normal computing time of ARIES scheme and shown in Figure 12. We can see that the normal computing time of DSIC mechanism is only 30 percent than that of ARIES mechanism on average since ARIES scheme creates and stores the latest data in every ADD/REMOVE operations. The normal computing time of DSIC-undo and DSIC-redo is a little bit longer than COMMAND scheme since COMMAND scheme only records the requested command in every operation instead of checking the validity of commands.

Figure 13 demonstrates the ratios of ADD operations to CHECKPOINT operations in ARIES, COMMAND, DSIC-undo, and DSIC-redo schemes. When $N$ is one, the ratios of checkpointing time to the total normal computing time of ARIES, COMMAND, DSIC-undo, and DSIC-redo schemes are 98, 5, 2 and 4 percent, respectively. When $N$ is one hundred, the ratios of checkpointing time to the total normal computing time of ARIES, COMMAND, DSIC-undo, and DSIC-redo schemes are 37, 0, 0 and 0 percent, respectively. The checkpointing time of ARIES scheme accounts for a large proportion. Thus, increasing the number of ADD operations between two checkpoints disperses the checkpointing time. On the other hand, the ratio of checkpointing time to the total normal computing time of COMMAND, DSIC-undo and DSIC-redo schemes is almost 0 percent since the checkpointing process only changes the state to the next version instead of replicating the whole data. That is, ADD operations dominate the normal computing time in COMMAND, DSIC-undo and DSIC-redo schemes.

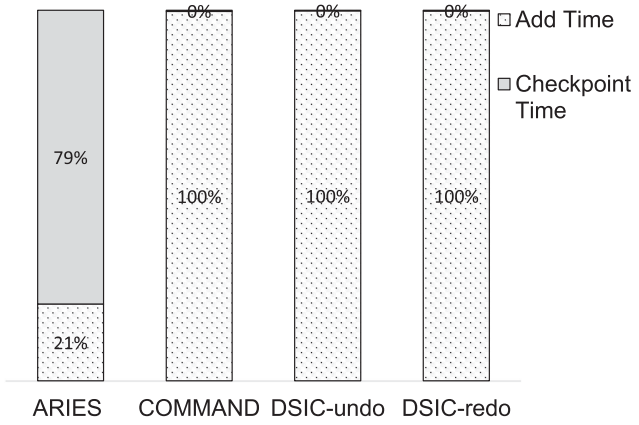
#### 2) NUMBER OF MEMBERS ADDED TO THE SET ($M$)

In this experiment, $M$ is set to one thousand, ten thousand and 0.1 million to measure the normal computing time. $Size_B$ is one million members in Redis, and $N$ is one hundred. As shown in Figure 14, when $M$ is 0.1 million members, the normal computing time of ARIES, COMMAND, DSIC-undo, and DSIC-redo schemes are 113 ms, 70 ms, 212 ms and 96 ms, respectively. When $M$ is one thousand, the normal computing time of ARIES, COMMAND, DSIC-undo, and DSIC-redo schemes are 4 ms, 0.77 ms, 2.6 ms, and 0.78 ms respectively. Since COMMAND and DISC-redo schemes do not need to modify the *base set*, the normal computing time of COMMAND and DISC-redo schemes is less than ARIES and DSIC-undo schemes. The smaller $M$, the greater the difference of the normal computing time among four approaches. Therefore, we can say that DSIC mechanism significantly reduces the normal computing time when a smaller number of members are added to the set.

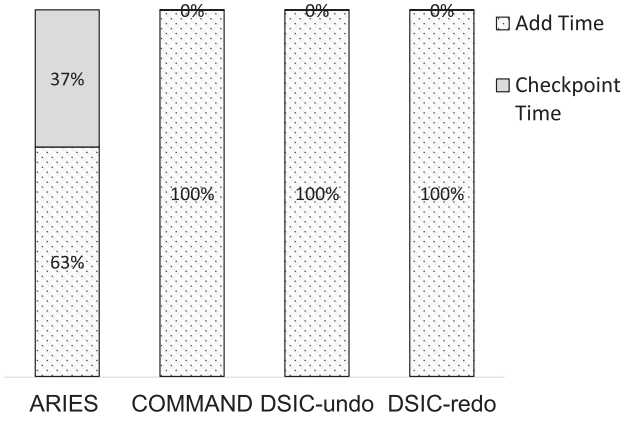
We also compare the storage cost for different $M$. We execute 50 different ADD operations and 50 same ADD operations. When $M$ is 0.1 million members, the storage sizes of ARIES, COMMAND, DSIC-undo, and DSIC-redo schemes are 1.31 GB, 2 GB, 1.62 GB, and 0.77 GB, respectively. When $M$ is one thousand members, the storage sizes of ARIES, COMMAND, DSIC-undo, and DSIC-redo schemes




(a) One add operation.




(b) Ten add operations.




(c) One hundred add operations.

**FIGURE 13.** **Ratio between checkpoint time and add time.**

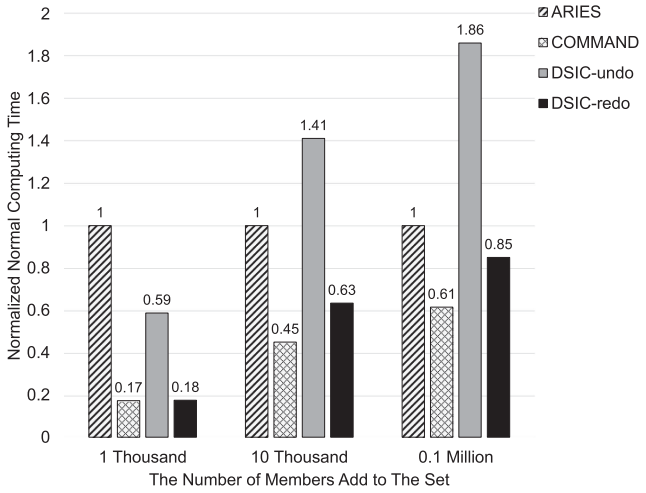



**FIGURE 14.** **Normalized normal computing time of different $M$.**

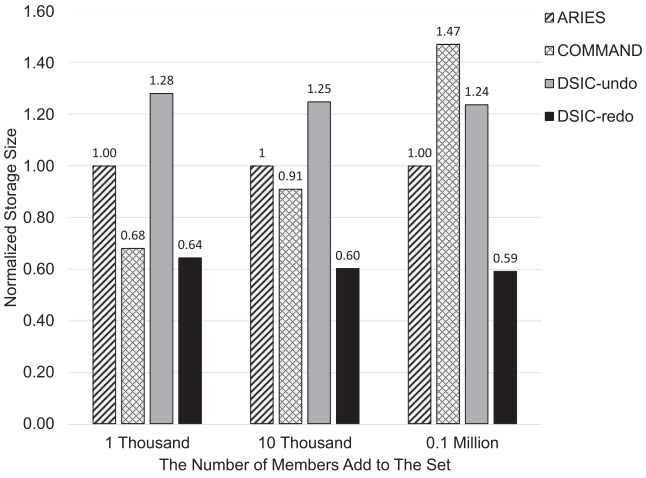



**FIGURE 15.** **Normalized storage size of different $M$.**

are 1.27 GB, 0.86 GB, 1.6 GB, and 0.8GB, respectively. The elements added to the set in the experiment are digits of only four bytes each. Thus, the total storage is less than 1.5 GB. We normalize the storage sizes of four approaches by the storage size of ARIES scheme in Figure 15. We can see that the storage cost of COMMAND scheme is highest since COMMAND scheme records every transaction and the added members in the log. When the add operation is repeated 50 times, COMMAND scheme records the same add operation 50 times. On the other hand, ARIES, DSIC-undo and DSIC-redo do not record repeated members. Compared to ARIES scheme, the proposed DISC mechanism reduces storage size by 40 percent. If the user adds larger elements, such as images to the set, DSIC mechanism can reduce more storage size. According to Figures 14 and 15, DSIC mechanism intensively saves the normal computing time and the storage cost.

### 3) SIZE OF THE *BASE SET* ($Size_B$)

$Size_B$ is set to 0.1 million members, one million members, and ten million members to measure the normal computing time in this experiment. $M$ is ten thousand members, and $N$ is ten. The normal computing time of ARIES scheme is used to normalize the normal computing time of four approaches in Figure 16. In this experiment, COMMAND scheme is least affected by the size of *base set* since COMMAND scheme only records transactions in every step. ARIES scheme is easily affected by the size of *base set*. When $Size_B$

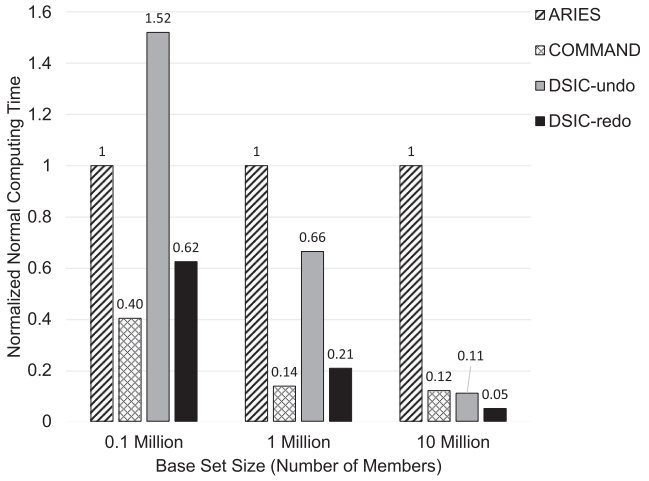



**FIGURE 16.** **Normalized normal computing time of different $Size_B$.**

is ten million members, the normal computing time of ARIES scheme is about 20 times that of DSIC mechanism. When $Size_B$ is one million members, the normal computing time of ARIES scheme is still longer than that of DSIC mechanism. ARIES scheme outperforms DSIC mechanism, but the difference is less than 10 ms with 0.1 million members in the *base set*. As the $Size_B$ increases, the normal computing time of DSIC mechanism is much shorter than that of the ARIES scheme. We can say that if the *base set* contains more than 0.1 million members, we should choose DSIC mechanism to reduce the normal computing time.

### B. EXPERIMENT OF READ OPERATIONS

In addition to ADD/REMOVE operations, READ operations are common. In this experiment, $Size_B$ is one million members in Redis. To evaluate the reading time between DSIC-redo and DSIC-undo schemes, the number of members in the *base set* and the *add set* should be large. Therefore, $N$ is one hundred and $M$ is set to 0.1 million to rapidly increase the number of members in the *base set*. The reading time is measured in the case of one, five, and ten checkpoints. Since the

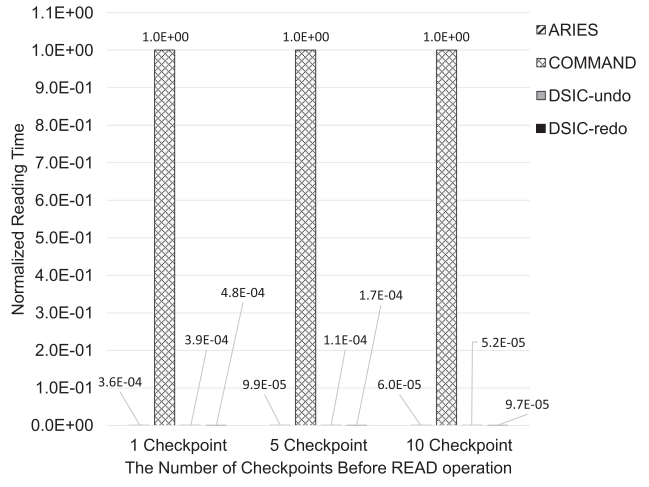



**FIGURE 17.** **Comparison of reading time.**

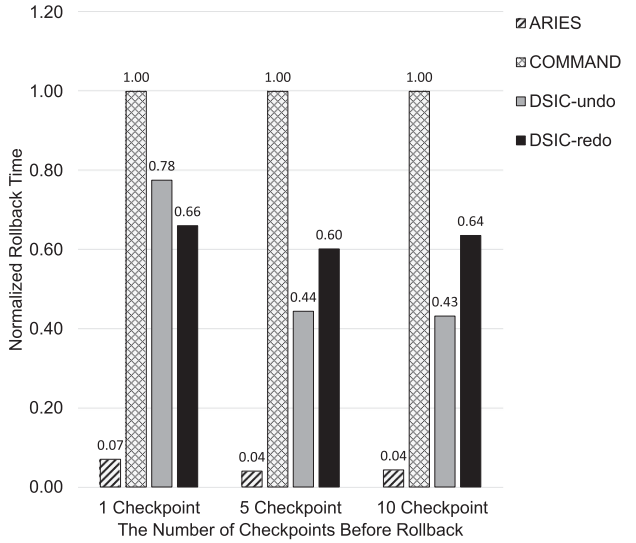



**FIGURE 18.** **Comparison of rollback time.**

reading time of COMMAND scheme is longest, we normalize the reading time of four approaches by the reading time of COMMAND scheme in Figure 17. The reading time of ARIES and DSIC-undo schemes are only 0.6 ms and 0.62 ms on average since the latest *base set* is maintained in ARIES and DSIC-undo schemes. The READ operation of DSIC-redo scheme searches for the specified element in the *add set* and *remove set* of each state. Although the READ operation of DSIC-redo scheme is more complicated than ARIES and DSIC-undo schemes, the reading time of DSIC-redo scheme is only 0.3 ms longer than that of ARIES and DSIC-undo schemes. On the other hand, COMMAND scheme executes all the transactions to read an element in the latest set and results in high computing complexity. The reading time of COMMAND scheme is 16590 times than that of ARIES, DSIC-undo and DSIC-redo schemes.

### C. EXPERIMENT OF ROLLBACK OPERATIONS

In the rollback experiment, $Size_B$ is one million members in Redis. $N$ is one hundred, and $M$ is 0.1 million members. The rollback time is measured in the case of one, five, and ten checkpoints. The rollback time of four approaches is normalized by the rollback time of COMMAND scheme. As shown in Figure 18, the rollback time of ARIES scheme is the shortest since the rollback procedure of ARIES scheme removes the latest *base set* without any other operations. COMMAND scheme operates all the logged transactions to finish the rollback operation and results in the longest rollback time. Compared to COMMAND scheme, DSIC-mechanism has much shorter rollback time since *add set* and the *remove set* are designed. In the rollback procedure of DSIC-undo scheme, we remove the value in the latest *add set* from the *base set* and add the value in the latest *remove set* to the *base set*. In DSIC-redo scheme, the rollback procedure combines the *add set* and the *remove set* of every state with the oldest *base set*. As shown in Figure 18, DSIC-undo and DSIC-redo can saves 50 percent rollback time that of COMMAND scheme.

**TABLE 2. Summary of experiments.**

| | ARIES | COMMAND | DSIC-undo | DSIC-redo |
|---|---|---|---|---|
| Suitable number of ADD/REMOVE operations between two checkpoints ($N$) | More than 100 | Any | Less than 100 | Any |
| Suitable number of members added to the set ($M$)-Computing Time | Fewer | Little effect | Fewer | Little effect |
| Suitable number of members added to the set ($M$)-Storage Cost | Fewer | Less repeated commands | Fewer | Little effect |
| Suitable *base set* size ($Size_B$) | Less than 0.1 million | More than 0.1 million | More than 0.1 million | More than 0.1 million |
| Rollback time | Shortest | Longest | Short | Short |
| Suitable application type | Failure-prone applications | No-failure applications | Read-intensive applications | Write-intensive applications |

### D. DISCUSSIONS

Table 2 summarizes the observations of our experiments. In DSIC schemes, the number of ADD/REMOVE operations and the number of members added to the set have no significant effect on the normal computing time. In ARIES scheme, more ADD/REMOVE operations between two checkpoints and a smaller number of members of an ADD/REMOVE operation disperses the normal computing time. COMMAND scheme has significant impact on storage cost, especially when there are many repeated transactions. If the *base set* size is larger than 0.1 million, COMMAND and DSIC mechanism achieves better performance. The rollback time of ARIES, DSIC-undo and DSIC-redo scheme is short. The rollback time of COMMAND scheme is too long and cannot be accepted by users. ARIES scheme is suitable for applications which crash frequently and often execute rollback procedures. DSIC-undo scheme performs better in read-intensive applications. DSIC-redo scheme yields shorter normal computing time in write-intensive applications. Although the rollback time of DSIC mechanism is longer than that of ARIES scheme, the checkpointing time of DSIC mechanism is much shorter than that of ARIES scheme. Therefore, DSIC mechanism can provide efficient checkpointing process to improve the reliability of streaming applications while retaining the ability to retrieve latest status and replay transactions.

## V. CONCLUSION

In this paper, we investigated the efficiency and the cost of fault tolerant techniques in stream processing systems. We utilized the order-irrelevant characteristics of sets to present a data-structure based incremental checkpointing (DSIC) mechanism for in-memory databases. In DSIC mechanism, we proposed DSIC-undo and DSIC-redo schemes. In these two schemes, the add, remove, read, checkpoint, and rollback processes were precisely designed and compared to ARIES logging method and command logging method.

From the experiment results, we observed that DSIC mechanism could achieve better performance when the size of the *base set* was greater than 0.1 million and there were less than one hundred ADD/REMOVE operations between two checkpoints. Under these conditions, our experiments showed that DSIC-undo and DSIC-redo schemes required less than 30 percent checkpointing time and a 40 percent reduction in storage space in comparison to ARIES scheme and COMMAND scheme. In addition, DSIC-undo and DSIC-redo schemes could be switched between each other and significantly reduced computational complexity for read-intensive and write-intensive streaming applications. In the future, we will further generalize the proposed incremental checkpointing mechanism for different data structures.

## ACKNOWLEDGMENTS

The authors sincerely thank Dr. Ku-Lung Wu and Dr. Gabriela Jacques da Silva in IBM T.J. Watson Research Center, Yorktown Heights, NY, USA, for their helps in defining this research problem and insightful discussions. This work was sponsored by the Ministry of Science and Technology (MOST) of Taiwan under grants MOST 109-2634-F-009-018 through Pervasive Artificial Intelligence Research (PAIR) Labs.

## REFERENCES

[1] J. Taylor, "Real-time responses with big data," 2014. [Online]. Available: https://blogs.oracle.com/rtd/real-time-responses-with-big-data

[2] M. A. U. Nasir, "Fault tolerance for stream processing engines," Tech. Rep., *arXiv:1605.00928*, 2016. [Online]. Available: https://arxiv.org/abs/1605.00928

[3] F. Zheng, C. Venkatramani, R. Wagle, and K. Schwan, "Cache topology aware mapping of stream processing applications onto cmps," in *Proc. IEEE Int. Conf. Distrib. Comput. Syst.*, 2013, pp. 52–61.

[4] Y. Kwon *et al.*, "Function-safe vehicular ai processor with nano core-in-memory architecture," in *Proc. Int. Conf. Artif. Intell. Circuits Syst.*, 2019, pp. 127–131.

[5] C.-H. Chen, F.-J. Hwang, and H.-Y. Kung, "Travel time prediction system based on data clustering for waste collection vehicles," *IEICE Trans. Info. Syst.*, vol. 102, no. 7, pp. 1374–1383, 2019.

[6] T. Shiobara, G. Habault, J.-M. Bonnin, and H. Nishi, "Effective communicating optimization for V2G with electric bus," in *Proc. IEEE Int. Conf. Ind. Inform.*, 2016, pp. 992–997.

[7] E. Pinheiro, W.-D. Weber, and L. A. Barroso, "Failure trends in a large disk drive population," in *Proc. USENIX Conf. File Storage Technol.*, 2007, pp. 17–29.

[8] B. Schroeder and G. Gibson, "A large-scale study of failures in high-performance computing systems," *IEEE Trans. Dependable Secure Comput.*, vol. 7, no. 4, pp. 337–350, Dec. 2010.

[9] X. Jiang, X. Cao, and D. H. Du, "Multihop transmission and retransmission measurement of real-time video streaming over DSRC devices," in *Proc. IEEE Int. Symp. World Wireless Mobile Multimedia Netw.*, 2014, pp. 1–9.

[10] X. Bao, L. Liu, N. Xiao, Y. Lu, and W. Cao, "Persistence and recovery for in-memory noSQL services: A measurement study," in *Proc. IEEE Int. Conf. Web Services*, 2016, pp. 530–537.
[11] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz, "Aries: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging," *ACM Trans. Database Syst.*, vol. 17, no. 1, pp. 94–162, 1992.
[12] N. Malviya, A. Weisberg, S. Madden, and M. Stonebraker, "Rethinking main memory OLTP recovery," in *Proc. IEEE Int. Conf. Data Eng.*, 2014, pp. 604–615.
[13] H. Zhang, G. Chen, B. C. Ooi, K.-L. Tan, and M. Zhang, "In-memory big data management and processing: A survey," *IEEE Trans. Knowl. Data Eng.*, vol. 27, no. 7, pp. 1920–1948, Jul. 2015.
[14] C. Yao, D. Agrawal, G. Chen, B. C. Ooi, and S. Wu, "Adaptive logging: Optimizing logging and recovery costs in distributed in-memory databases," in *Proc. ACM Int. Conf. Manage. Data*, 2016, pp. 1119–1134.
[15] M. Balazinska, H. Balakrishnan, S. Madden, and M. Stonebraker, "Fault-tolerance in the borealis distributed stream processing system," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2005, Art. no. 3.
[16] E. N. Elnozahy, L. Alvisi, Y.-M. Wang, and D. B. Johnson, "A survey of rollback-recovery protocols in message-passing systems," *ACM Comput. Surv.*, vol. 34, no. 3, pp. 375–408, 2002.
[17] Y. Kwon, M. Balazinska, and A. Greenberg, "Fault-tolerant stream processing using a distributed, replicated file system," *Proc. VLDB Endowment*, vol. 1, no. 1, pp. 574–585, 2008.
[18] J. Zhou, C. Zhang, H. Tang, J. Wu, and T. Yang, "Programming support and adaptive checkpointing for high-throughput data services with log-based recovery," in *Proc. IEEE Int. Conf. Dependable Syst. Netw.*, 2010, pp. 91–100.
[19] Z. Sebepou and K. Magoutis, "CEC: Continuous eventual checkpointing for data stream processing operators," in *Proc. IEEE Int. Conf. Dependable Syst. Netw.*, 2011, pp. 145–156.
[20] R. Castro Fernandez, M. Migliavacca, E. Kalyvianaki, and P. Pietzuch, "Integrating scale out and fault tolerance in stream processing using operator state management," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2013, pp. 725–736.
[21] S. Nathan and B. Gedik, "Using infosphere streams with memcached and redis," 2017. [Online]. Available: https://www.ibm.com/developerworks/library/bd-streamsmemcached/
[22] Redis Labs, "Redis," 2017. [Online]. Available: http://redis.io/
[23] J. L. Carlson, *Redis in action*. Shelter Island, NY, USA: Manning Publications Co., 2013.
[24] B. Fitzpatrick, "Memcached," 2018. [Online]. Available: https://memcached.org/
[25] D. Merriman, E. Horowitz, and K. Ryan, "Mongodb," 2018. [Online]. Available: https://www.mongodb.com/
[26] D. Matar, "Benchmarking fault-tolerance in stream processing systems," Masters thesis, Dept. Comput. Sci., TU-Berlin, 2016.
[27] D. Min, T. Hwang, J. Jang, Y. Cho, and J. Hong, "An efficient backup-recovery technique to process large data in distributed key-value store," in *Proc. ACM Symp. Appl. Comput.*, 2015, pp. 2072–2074.
[28] S. Chen, X. Tang, H. Wang, H. Zhao, and M. Guo, "Towards scalable and reliable in-memory storage system: A case study with redis," in *Proc. IEEE Int. Conf. Big Data Sci. Eng.*, 2016, pp. 1660–1667.
[29] F. Renkes and J.-H. Bose, "Logging scheme for column-oriented in-memory databases," US Patent 8,868,512, Oct. 21, 2014.
[30] M. M. Yiu, H. H. Chan, and P. P. Lee, "Erasure coding for small objects in in-memory KV storage," in *Proc. ACM Int. Syst. Storage Conf.*, 2017, pp. 1–12.
[31] Redis Labs, "Redis persistance," 2012. [Online]. Available: https://redis.io/topics/persistence
[32] E. Giles, K. Doshi, and P. Varman, "Persisting in-memory databases using SCM," in *Proc. IEEE Int. Conf. Big Data*, 2016, pp. 2981–2990.
[33] P. Damani and R. E. Strom, "System and method for maintaining checkpoints of a keyed data structure using a sequential log," US Patent 7 451 166, Nov. 11, 2008.
[34] J. L. Lawall and G. Muller, "Efficient incremental checkpointing of java programs," in *Proc. IEEE Int. Conf. Dependable Syst. Netw.*, 2000, pp. 61–70.
[35] J. S. Plank, M. Beck, G. Kingsley, and K. Li, "Libckpt: Transparent checkpointing under unix," in *Proc. Usenix Techn. Conf.*, 1995, pp. 213–223.

**CHIA-YU LIN** (Member, IEEE) received the BS and MS degrees in computer science from National Chiao Tung University (NCTU), Taiwan in 2010 and 2012, respectively, and the PhD degree from the Institute of Communications Engineering, NCTU, in 2019. She is the researcher in NCTU. Her current research interests include real-time updating techniques for recommendation algorithms and mathematical framework for data streaming applications.

**LI-CHUN WANG** (Fellow, IEEE) received the BS degree in electrical engineering from National Chiao Tung University, Taiwan, in 1986, the MS degree in electrical engineering from National Taiwan University, in 1988, and the MS and PhD degrees in electrical engineering from the Georgia Institute of Technology, Atlanta, in 1995 and 1996, respectively. From 1990 to 1992, he was with the Telecommunications Laboratories of the Ministry of Transportation and Communications, Taiwan. In 1995, he was affiliated with the Bell Northern Research of Northern Telecom, Inc., Richardson, TX. From 1996 to 2000, he was with AT&T Laboratories, where he was a senior technical staff member with the Wireless Communications Research Department. Since August 2000, he has been with National Chiao Tung University, Taiwan, where he is currently a chair professor with the Department of Electrical and Computer Engineering and is jointly appointed by the Department of Computer Science. He has published more than 90 journal papers and 180 conference papers, 19 U.S. patents, and has co-edited a book entitled *Key Technologies for 5G Wireless Systems* (Cambridge, 2017). His recent research interests include cross-layer optimization and data-driven learning techniques for 5G ultra-reliable and ultra-low latency communications, edge computing, unmanned aerial vehicle communications networks, and AI-empowered mobile networks. He was a recipient of the 1997 IEEE Jack Neubauer Best Paper Award in 1997, the Distinguished Research Award of Ministry of Science and Technology, Taiwan, in 2012 and 2018, and the IEEE Communications Society Asia–Pacific Board Best Paper Award in 2015. He was elected as an IEEE fellow for his contributions in cellular architectures and radio resource management in wireless networks.

**SHU-PING CHANG** (Member, IEEE) received the PhD degree in computer and information sciences from the University of Minnesota with special focus in computer communication and system. He works at AI Engineering, T.J. Watson Research Center as a software development manager for the IBM System S (Streams) Laboratory, a cluster for distributed computing research and development. He has more than 25 years research and product development experiences in the computer and information technology arena. He has broad and in depth knowledge in computer system hardware architecture and software structure in big data platforms and prototype building and development, computer communication, relational database, Internet based solutions, and cloud computing.